[PSI5790-2025 aula 3 parte 1 – início]

Transformações geométricas

As imagens usadas nesta aula podem ser baixadas em Linux com os comandos:

```
$ wget -U 'Firefox/50.0' http://www.lps.usp.br/hae/apostila/transformacao.zip
$ unzip transformacao.zip
```

Nota: O arquivo "procimagem.h" está em http://www.lps.usp.br/hae/apostila/procimagem.h.

1. Mudança de escala e interpolação

1.1 Introdução

Considere o problema de aumentar a resolução de uma imagem 3×3 pixels (imagem azul na figura 1) para 4×4 pixels (imagem vermelha). Neste problema, podemos considerar que:

(a) A imagem é constituída de pixels que são pequenos quadrados (figura 1a) e o nível de cinza é constante dentro de cada pixel; ou

(b) A imagem é uma função contínua no domínio 2D que foi amostrada em certos pontos (figura 1b).

Dependendo da técnica estudada, fica mais fácil resolvê-la imaginando o problema de uma forma ou de outra.

Para aumentar a resolução da imagem, é melhor fazer a suposição *b* (imaginar pixels como pontos onde a função foi amostrada). Assim, o problema de aumentar resolução torna-se estimar as cores da imagem nos círculos vermelhos, conhecendo as cores nos triângulos azuis (figura 1b).

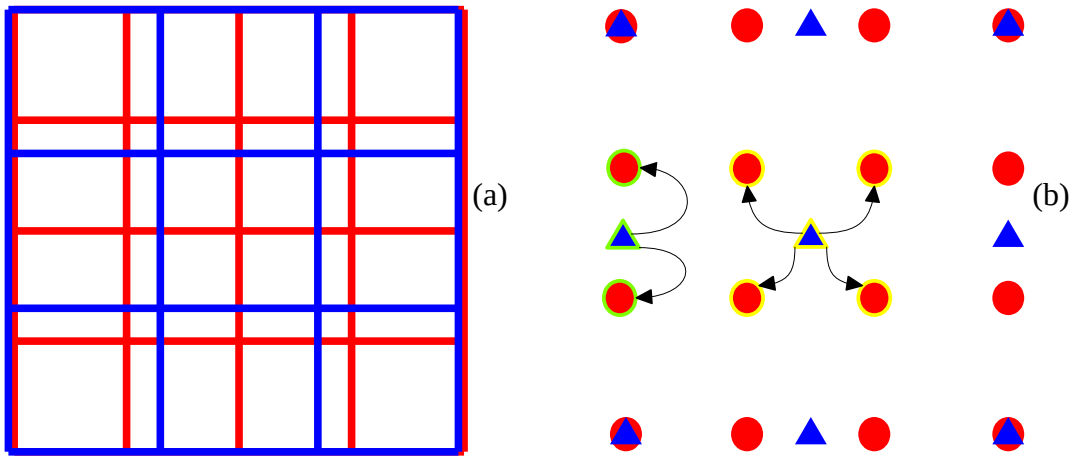


Figura 1: Na reamostragem, podemos imaginar que: (a) a imagem é constituída de pixels quadrados ou; (b) a imagem é uma função contínua que foi amostrada em alguns pontos.

A figura Y mostra a interpolação interpretada desta forma. A função *f* da figura representa uma imagem em níveis de cinza. Interpolar é estimar o valor da *f* num certo ponto do domínio (por exemplo, no pontinho verde) conhecendo o valor de *f* somente nas intersecções da grade.

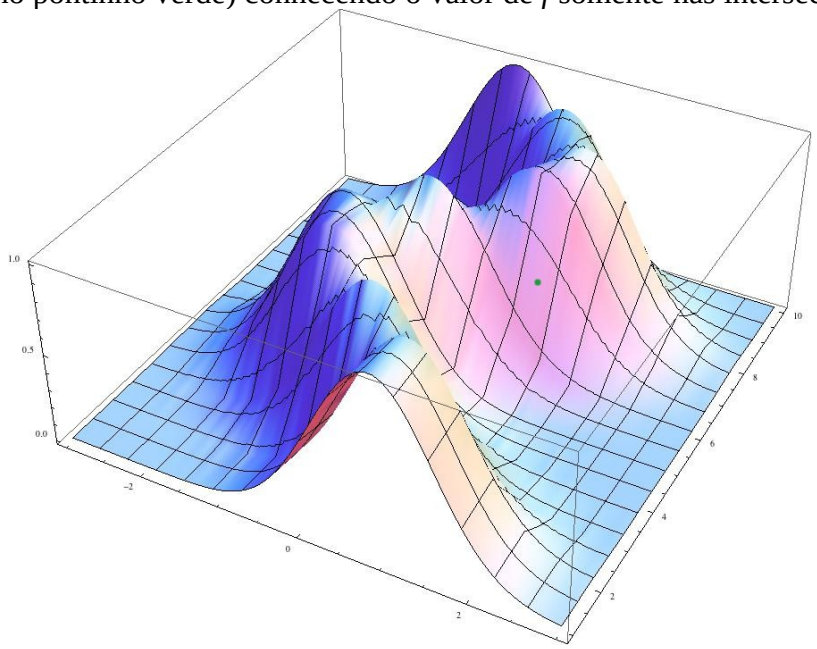


Figura Y: Interpolar é estimar o valor da função num certo ponto do domínio (por exemplo, na bolinha verde) conhendo os valores da função somente nas intersecções da grade.

1.2 Interpolação vizinho mais próximo

A solução mais simples para este problema é a interpolação vizinho mais próximo. Nesta solução, cada pixel da imagem de saída (um círculo vermelho na figura 1b) recebe a cor do pixel espacialmente mais próximo na imagem de entrada (triângulo azul). Assim, os 2 círculos vermelhos com contornos verdes receberão a cor do triângulo azul com contorno verde. E os 4 círculos vermelhos com contornos amarelos receberão a cor do triângulo azul com contorno amarelo. Assim, todos os 4 pixels de saída com contorno amarelo terão a mesma cor. O fato de vários pixels de saída terem a mesma cor irá gerar o efeito indesejável de "blocos com cor uniforme" ou "escadinha". Este algoritmo pode ser implementado especificando:

a) Os fatores de ampliação de linha e coluna (*fatorl* e *fatorc*); ou
b) Os números de linhas e colunas (*nl* e *nc*) da imagem de saída.

O programa 1 implementa a interpolação vizinho mais próximo usando a especificação (*a*). As linhas 10-13 fazem a leitura da imagem de entrada e dos parâmetros de ampliação *fatorl* e *fatorc*. As linhas 14-15 calculam os números de linhas e colunas da imagem de saída *b*. A função *cvRound* de OpenCV arredonda a variável ponto flutuante diretamente para o número inteiro (a função *round* de C/C++ arredonda variável ponto flutuante para número ponto flutuante e portanto é menos eficiente). A linha 16 cria imagem de saída com *nl*×*nc* pixels. As linhas 17-19 percorrem todos os pixels da imagem de saída *b*, calculando as coordenadas do pixel da imagem de entrada *a* mais próxima. Depois, copia o seu valor para imagem de saída.

```cpp
// vizinhocv.cpp - 2024
// Especifica fatores de ampliacao
#include "procimagem.h"
int main(int argc, char** argv)
{ if (argc!=5) {
    printf("vizinho: Muda resolucao de imagem usando interpolacao vizinho+px.\n");
    printf("vizinho ent.pgm sai.pgm fatorl fatorc\n");
    erro("Erro: Numero de argumentos invalido");
  }
  Mat_<uchar> a=imread(argv[1],0); if (a.total()==0) erro("Erro leitura");
  float fatorl,fatorc;
  if (sscanf(argv[3],"%f",&fatorl)!=1) erro("Erro: Leitura fatorl");
  if (sscanf(argv[4],"%f",&fatorc)!=1) erro("Erro: Leitura fatorc");
  int nl=round(a.rows*fatorl);
  int nc=round(a.cols*fatorc);
  Mat_<uchar> b(nl,nc);
  for (int l=0; l<b.rows; l++)
    for (int c=0; c<b.cols; c++)
      b(l,c) = a(round(l/fatorl),round(c/fatorc));
  imwrite(argv[2],b);
}
```

```python
# vizinho.py - pos2022
# Especifica fatores de ampliacao

import cv2, sys
import numpy as np
if len(sys.argv)!=5:
  print("vizinho: Muda resolucao de imagem usando interpolacao vizinho+px.")
  print("vizinho ent.pgm sai.pgm fatorl fatorc")
  sys.exit("Erro: Numero de argumentos invalido")

a=cv2.imread(sys.argv[1],0)
if a is None:
  sys.exit("Erro leitura"+sys.argv[1])
fatorl=float(sys.argv[3]); fatorc=float(sys.argv[4])
nl=round(a.shape[0]*fatorl); nc=round(a.shape[1]*fatorc)
b=np.empty((nl,nc),np.uint8)
for l in range(b.shape[0]):
  for c in range(b.shape[1]):
    b[l,c] = a[int(l/fatorl),int(c/fatorc)]
cv2.imwrite(sys.argv[2],b)
```

Programa 1: Mudança de escala da imagem usando interpolação vizinho mais próximo, em C++/OpenCV e Python/OpenCV.

Executando:

```
>vizinho lennag-reduz.jpg vizinho-ampl.jpg 1.2 1.8
>vizinho lennag-reduz.jpg vizinho-reduz.jpg 0.8 0.6
```

obtemos as saídas mostradas na figura 2.

Figura 2: Ampliação e redução usando interpolação vizinho mais próximo.

Exercício: Modifique o programa 1 (vizinho.cpp) para receber como parâmetro os números de linhas e colunas da imagem de saída (especificação *b*).

1.3 Interpolação bilinear

Como previmos, a imagem “vizinho-ampl.jpg” apresenta “escadinhas” que diminuem a qualidade visual da imagem (visível, por exemplo, na fronteira da bochecha com os cabelos). Podemos eliminar essas “escadinhas” ou “bloquinhos” substituindo a interpolação vizinho mais próximo por alguma interpolação mais sofisticada, por exemplo, a bilinear. A interpolação bilinear tira a média aritmética ponderada dos 4 pixels de entrada que circundam o pixel de saída (figura 3). Com isso, não haverá mais “bloquinhos de cor uniforme”.

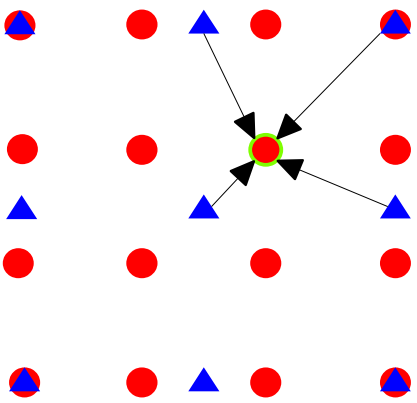
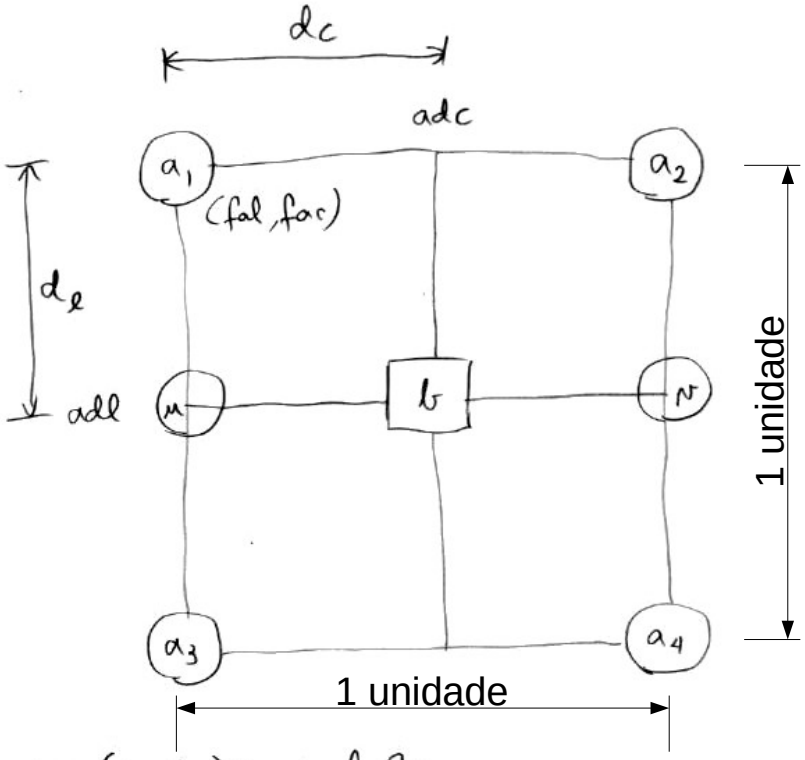


Figura 3: Interpolação bilinear calcula média aritmética ponderada das cores dos 4 pixels de entrada que circundam o pixel de saída.



$$\mu = (1-d_l)a_1 + d_l a_3$$

$$\nu = (1-d_l)a_2 + d_l a_4$$

$$b = (1-d_c)\mu + d_c \nu$$

$$b = (1-d_c)\left[(1-d_l)a_1 + d_l a_3\right] + d_c\left[(1-d_l)a_2 + d_l a_4\right]$$

$$b = (1-d_c)(1-d_l)a_1 + (1-d_c)d_l a_3 + d_c(1-d_l)a_2 + d_c d_l a_4$$

Figura 4: Equação para interpolação bilinear.

Figura 4 mostra as equações para fazer interpolação bilinear. Suponha que o pixel de saída b esteja circundado pelos pixels de entrada a_1, a_2, a_3 e a_4, como mostra a figura 4. Vamos supor que a distância entre dois pixels consecutivos é 1 unidade e a distância a_1 até b é d_c horizontalmente e d_l verticalmente. Neste caso, a cor de b pode ser calculada como média ponderada:

$$b = p_1 a_1 + p_2 a_2 + p_3 a_3 + p_4 a_4 \quad ,$$

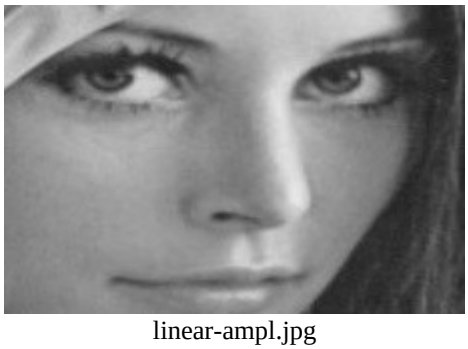
onde:

$$\begin{cases} p_1 = (1-d_c)(1-d_l) \\ p_2 = d_c(1-d_l) \\ p_3 = (1-d_c)d_l \\ p_4 = d_c d_l \end{cases}$$

```
 1 //linear.cpp - 2024
 2 #include "procimagem.h"
 3 int main(int argc, char** argv) {
 4   if (argc!=5) {
 5     printf("linear: Muda resolucao de imagem usando interpolacao bilinear.\n");
 6     printf("linear ent.pgm sai.pgm nl nc\n");
 7     erro("Erro: Numero de argumentos invalido");
 8   }
 9   Mat_<uchar> a=imread(argv[1],0); if (a.total()==0) erro("Erro leitura");
10   int nl,nc;
11   if (sscanf(argv[3],"%d",&nl)!=1) erro("Erro: Leitura nl");
12   if (sscanf(argv[4],"%d",&nc)!=1) erro("Erro: Leitura nc");
13   Mat_<uchar> b(nl,nc);
14   for (int l=0; l<b.rows; l++)
15     for (int c=0; c<b.cols; c++) {
16       double ald = l * ((a.rows-1.0)/(b.rows-1.0));
17       double acd = c * ((a.cols-1.0)/(b.cols-1.0));
18       int fal=int(ald); int fac=int(acd);
19       double dl=ald-fal; double dc=acd-fac;
20
21       double p1=(1-dl)*(1-dc);
22       double p2=(1-dl)*dc;
23       double p3=dl*(1-dc);
24       double p4=dl*dc;
25       b(l,c)= cvRound(
26                 p1*a(fal,fac)   + p2*a(fal,fac+1)  +
27                 p3*a(fal+1,fac) + p4*a(fal+1,fac+1)
28               );
29     }
30   imwrite(argv[2],b);
31 }
```

Programa 2: Mudança de escala da imagem usando interpolação bilinear.

imagem de entrada
lenna-reduz.jpg



linear-ampl.jpg

linear-reduz.jpg

Figura 5: Ampliação e redução usando interpolação bilinear.

O programa 2 implementa a mudança de escala da imagem usando interpolação bilinear. Normalmente, os programas de transformação geométrica percorrem os pixels da imagem de saída, procurando calcular os seus valores. No programa 2 (linear.cpp), as linhas 13-29 percorrem os pixels da imagem de saída *b*. As linhas 16-19 calculam as variáveis d_l e d_c das equações da figura 4. As linhas 21-24 calculam os 4 pesos e as linhas 25-28 calculam a média ponderada e armazena o resultado na imagem de saída.

Executando:

```
>linear lennag-reduz.jpg linear-ampl.jpg 154 230
>linear lennag-reduz.jpg linear-reduz.jpg 102 77
```

obtemos as saídas mostradas na figura 5. Compare a figura 5 com figura 2 e veja como a interpolação bilinear diminuiu o efeito “escadinha” (por exemplo, na bochecha). A figura 6 mostra mais claramente as diferenças entre interpolações vizinho mais próximo e bilinear.

vizinho mais próximo

bilinear

Figura 6: Detalhe da imagem lennag.jpg (512×512) reamostrada para 600×700 pixels usando interpolações vizinho mais próximo e bilinear.

Exercício: Execute programas 1 e 2 para ampliar bastante uma imagem de sua escolha e verifique a diferença de qualidade entre as duas saídas.

1.4 Função resize de OpenCV

Por motivos didáticos, escrevemos manualmente as rotinas de mudança de escala das imagens. Porém, OpenCV possui a função pronta *resize* que faz essa tarefa. A sua sintaxe é:

C++:
```
void resize(InputArray src, OutputArray dst, Size dsize,
            double fx=0, double fy=0, int interpolation=INTER_LINEAR)

Exemplo para mudar escala por fator usando interpolação viz+px:
  resize(a, b, Size(0,0), fatorx, fatory, INTER_NEAREST);
Exemplo para mudar o tamanho para 512×512:
  resize(a, b, Size(512,512));
```

Python:
```
cv2.resize(src, dsize[, dst[, fx[, fy[, interpolation ] ] ] ]) → dst

Exemplo:
      b=cv2.resize(a, (512,512))
```

Os métodos de interpolação disponíveis são:

```
INTER_NEAREST   nearest-neighbor interpolation 1x1
INTER_LINEAR    bilinear interpolation (used by default) 2x2
INTER_CUBIC     bicubic interpolation over 4x4 pixel neighborhood
INTER_LANCZOS4  Lanczos interpolation over 8x8 pixel neighborhood
INTER_AREA  resampling using pixel area relation. It may be the preferred
            method for image decimation, as it gives moire-free results.
            But when the image is zoomed, it is similar to
            the INTER_NEAREST method.
```

Exemplo:

```
 1 //cvvizinho.cpp 2024
 2 #include "procimagem.h"
 3 int main(int argc, char** argv) {
 4   if (argc!=4) {
 5     printf("cvvizinho ent.pgm sai.pgm fator\n");
 6     erro("Erro: Numero de argumentos invalido");
 7   }
 8   Mat_<uchar> a=imread(argv[1],0); if (a.total()==0) erro("Erro leitura");
 9   double fator; sscanf(argv[3],"%lf",&fator); //testar
10   Mat_<uchar> b;
11   resize(a, b, Size(0,0), fator, fator, INTER_NEAREST);
12   imwrite(argv[2],b);
13 }
```

Programa 3: Mudança de escala da imagem usando função *resize* do OpenCV.

1.5 Interpolações bicúbica e Lanczos

Estudamos até aqui interpolações vizinho mais próximo e bilinear que levam em consideração respectivamente vizinhanças 1×1 e 2×2. As interpolações bicúbica e Lanczos levam em consideração vizinhanças 4×4 e 8×8. A figura 7 mostra graficamente as interpolações vizinho mais próximo, linear e cúbico num sinal 1D e numa imagem 2D.

As diferenças entre essas interpolações se tornam mais evidentes quando ampliamos bastante a imagem. A figura 8 mostra a imagem *lennag.jpg* com 512×512 pixels ampliada 320% usando diferentes interpolações. Note como a qualidade da imagem vai aumentando.

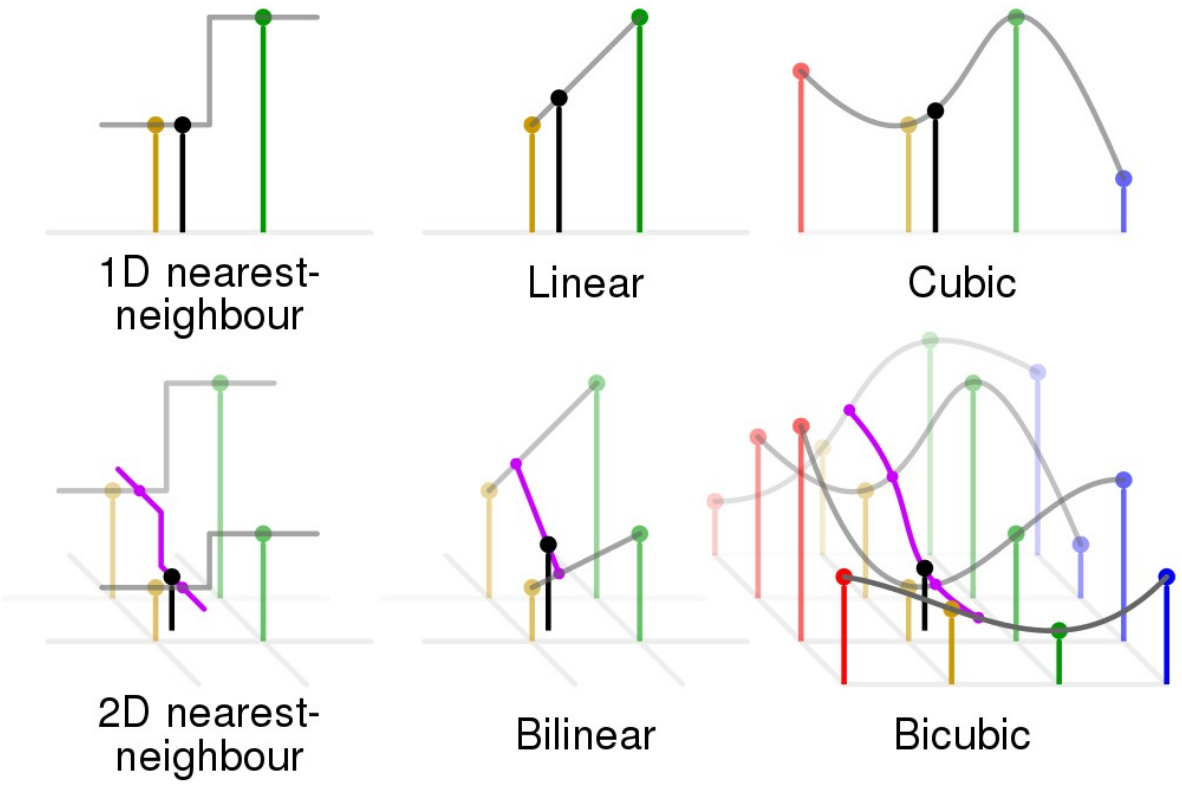


Figura 7: Interpolação bicúbica (retirado de [wikiBicubic]).

Exercício: Escreva um programa que redimensiona a imagem lennag.jpg usando a função *resize* de OpenCV e amplie bastante (360%) essa imagem usando os 4 métodos de interpolação (vizinho, bilinear, bicúbico e Lanczos). Verifique a diferença de qualidade das 4 saídas.

vizinho mais próximo

bilinear

bicúbico

Lanczos4

Figura 8: Detalhe da imagem lennag.jpg (512×512) ampliada 320% usando diferentes interpolações.

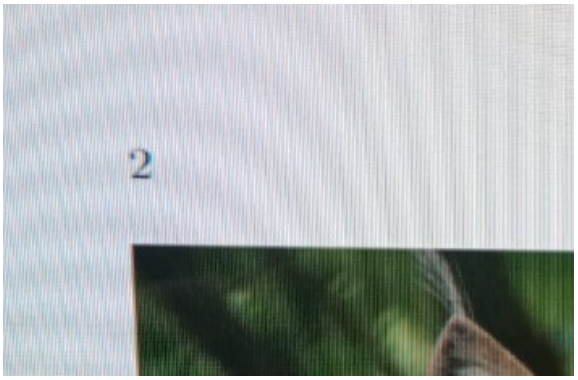
1.6 Redução de imagem

As quatro interpolações que vimos acima servem para fazer transformações geométricas que aumentam a resolução da imagem (ou que mantém aproximadamente a resolução original, como a rotação). Porém, são inadequadas para reduzir a resolução da imagem, pois todos eles podem gerar *aliasing* ou padrão de *Moiré* [wikiAliasing]. Figuras F1 e 9 mostram exemplos de *aliasing*.

Nota: Em áudio, a sobreposição de duas frequências próximas que gera uma terceira frequência é denominada de batimento.

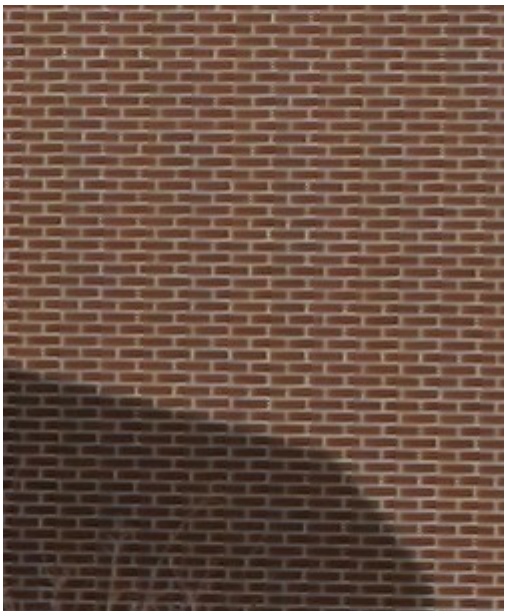
Ao reduzir a imagem, pode ser que numa região sejam amostradas majoritariamente a cor de tijolo, enquanto numa outra região podem ser amostradas majoritariamente a cor do cimento. Se uma região da imagem tiver sido amostrada como na figura 9f, essa região terá somente cor de tijolo. Pode ser que numa outra região seja amostrada somente a cor do cimento. Assim, na figura 9b aparecem alternadamente faixas com cor de tijolo e de cimento.

(a) Batimento entre as listras da camisa e os pixels da tela do computador cria faixas escuras e claras.



(b) Batimento entre os pixels do monitor e da câmera cria faixas.

Figura F1: Exemplos de *aliasing* ou *padrão de Moiré* (figuras de [https://www.intmath.com/math-art-code/moire-effect.php] e [Qualificação_Guilherme]).



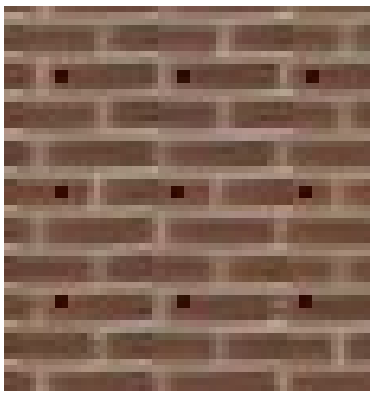
(a) Parte da imagem original (parede_de_tijolos.png).

(b) Redução para escala 0.4 pelo vizinho mais próximo, sem filtro passa-baixa cria listas.

(c) Redução para escala 0.4 com filtro gaussiano de desvio 0.83 pixels (2*downscale/6).

(d) Redução para escala 0.4 com filtro gaussiano de desvio 1.67 pixels (4*downscale/6).

(e) Redução para escala 0.4 com INTER_AREA.



(f)

Figura 9: Exemplo de aliasing (de [wikiAliasing]). Quando a imagem (a) é reduzida para escala 0,4 pela interpolação vizinho mais próximo, aparece aliasing (b). Passando filtro gaussiano, aliasing é atenuada (c-d). Usando interpolação por área (e) aliasing é praticamente imperceptível.

Para evitar *aliasing*, costuma-se aplicar algum filtro passa-baixa (normalmente o filtro gaussiano) antes de diminuir a resolução da imagem (normalmente pela interpolação vizinho mais próximo). O filtro passa-baixa faz com que, em vez de amostrar um único pixel, a média de cores da região seja amostrada.

Nota: A biblioteca Scikit aplica o filtro gaussiano com desvio-padrão σ=2*downscale/6 [scikitReduce] para reduzir imagem ao gerar estrutura multi-escala piramidal (figura 9c). A biblioteca OpenCV também aplica filtro gaussiano antes de reduzir a imagem para gerar estrutura piramidal [opencvPyramids].

A biblioteca OpenCV oferece a interpolação *inter_area* projetada especialmente para reduzir a imagem (figura 9e):

```
INTER_AREA  resampling using pixel area relation. It may be the preferred
            method for image decimation, as it gives moire-free results.
            But when the image is zoomed, it is similar to
            the INTER_NEAREST method.
```

Na figura 10, vamos supor que a imagem de entrada é vermelha e queremos reduzir a sua resolução, obtendo a imagem azul. Para calcular a cor do pixel de saída *P*, a interpolação *inter_area* da OpenCV provavelmente calcula a média aritmética dos pixels *A*, *B*, *C* e *D* da imagem de entrada com pesos proporcionais às áreas das intersecções de *A*, *B*, *C* e *D* com o pixel *P* [https://medium.com/@wenrudong/what-is-opencvs-inter-area-actually-doing-282a626a09b3].

Nota: Não tenho certeza de que este é exatamente o o algoritmo do OpenCV, pois o manual não traz mais informações.

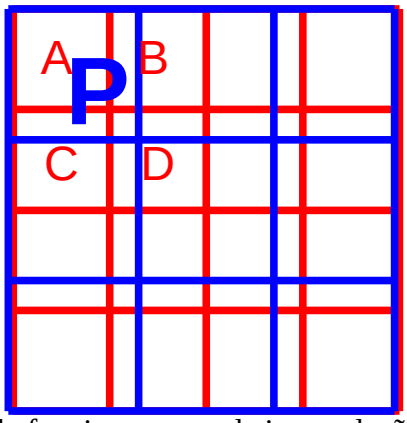


Figura 10: Provável modo de funcionamento da interpolação “inter_area” de OpenCV.

Nota: Gimp disponibiliza as reamostragens NoHalo e LoHalo para reduzir imagem.
https://graphicdesign.stackexchange.com/questions/138059/what-algorithm-do-the-nohalo-and-lohalo-resize-methods-use-in-gimp

Exercício: Explique por que a interpolação vizinho mais próximo pode gerar aliasing quando reduz a imagem.

Exercício: Explique por que aplicar filtro gaussiano (usando desvio-padrão apropriado) antes de fazer interpolação reduz *aliasing*.

Exercício: Escreva um programa que faz redução de imagem usando as 4 interpolações (vizinho, linear, cúbico e Lanczos).

Exercício: Escreva um programa que efetua as seguintes reduções para escala 0.3 (downscale 3.333) da imagem “parede_de_tijolos.png”:
a) Redução por vizinho mais próximo sem filtro.
b) Redução por vizinho mais próximo com filtro gaussiano de desvio-padrão 2*downscale/6 pixels.
c) Redução por vizinho mais próximo com filtro gaussiano de desvio-padrão 4*downscale/6 pixels
d) Redução por interpolação INTER_AREA do OpenCV.

Nota: Cada programa parece fazer uma operação diferente quando se reduz imagem usando interpolações bilinear e bicúbico.

2. Rotação

2.1 Equações

Para rotacionar uma imagem *A* resultando na imagem rotacionada *B*, é necessário saber calcular, para cada pixel (x_B, y_B) da imagem de saída *B*, a coordenada correspondente (x_A, y_A) da entrada *A*. Depois disso, podemos aplicar as técnicas de interpolação que já vimos acima. A fórmula de rotação no sistema de coordenadas cartesiano está na figura 11.

$$c = \cos(\theta) \qquad s = \text{sen}(\theta)$$

$$[x_B, y_B] = [x_A, y_A] \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \quad (1)$$

$$[x_A, y_A] = [x_B, y_B] \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \quad (2)$$

onde θ é o ângulo de rotação. Ou equivalentemente:

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} x_A \\ y_A \end{bmatrix} \quad (1)$$

$$\begin{bmatrix} x_A \\ y_A \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x_B \\ y_B \end{bmatrix} \quad (2)$$

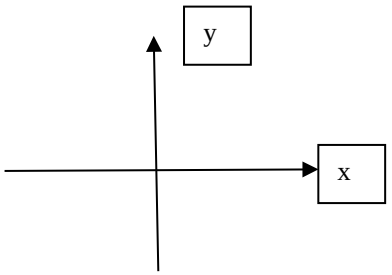


Figura 11: Equações para rotação em torno do centro de sistema de coordenadas. (x_A, y_A) são as coordenadas do pixel antes da rotação e (x_B, y_B) são as suas coordenadas após a rotação.

A equação (1) da figura 11 é a transformação direta, que leva as coordenadas de *A* em *B*. A equação (2) é a transformação inversa, que leva coordenadas de *B* em *A*. As duas matrizes são uma inversa da outra. Na prática, precisamos da equação (2), que leva *B* em *A*, para fazer rotação, pois precisamos calcular, para cada pixel de saída, as coordenadas do pixel correspondente na entrada. Temos três problemas para aplicar a equação (2) para rotacionar uma imagem:

1. As coordenadas dos pixels em OpenCV estão na forma matricial (*l*, *c*) e não em coordenadas cartesianas (*x*, *y*) da equação.
2. A equação (2) acima faz girar os pixels em torno do centro do sistema de coordenadas, isto é, a imagem rotaciona em torno do canto superior esquerdo e não em torno do centro da imagem.
3. Eventualmente, a equação (2) pode levar a um pixel fora do domínio da imagem de entrada *A*. Assim, o programa teria que ficar testando se (x_A, y_A) pertence ou não ao domínio da imagem *A*.

2.2 Classe ImgXyb

Certamente, é possível resolver todos os problemas acima manipulando as fórmulas matematicamente (apesar de que dá um certo trabalho) e fazendo testes para verificar se um pixel está dentro ou fora do domínio. Em vez disso, podemos criar uma classe *ImgXyb* que permite trabalhar diretamente com sistema de coordenadas cartesiano e sem nos preocuparmos se estamos acessando pixel dentro do domínio.

```
//procimagem.h 2024
(...)

template<typename T> class ImgXyb: public Mat_<T> {
public:
  using Mat_<T>::Mat_; //inherit constructors

  T backg;
  int lc=0, cc=0;
  int minx=0, maxx=this->cols-1, miny=1-this->rows, maxy=0;

  void centro(int _lc, int _cc) {
    lc=_lc; cc=_cc;
    minx=-cc; maxx=this->cols-cc-1;
    miny=-(this->rows-lc-1); maxy=lc;
  }

  T& operator()(int x, int y) { // modo XY centralizado
    unsigned li=lc-y; unsigned ci=cc+x;
    if (li<unsigned(this->rows) && ci<unsigned(this->cols))
      return (*this).Mat_<T>::operator()(li,ci);
    else return backg;
  }
};
```

Programa P: Classe *ImgXyb* simplificada inserida no *procimagem.h*.

A classe *ImgXyb* é derivada da classe *Mat_*, de forma que *ImgXyb* pode ser usada em praticamente todos lugares onde *Mat_* é usada. Porém, possui:

a) O método *centro*(*l*,*c*) que permite indicar onde está o centro do sistema de coordenadas;
b) O membro *backg* que especifica qual é a cor que a classe irá retornar quando acessar um pixel fora do seu domínio.
c) Os membros *minx*, *maxx*, *miny* e *maxy* que indicam os limites do domínio da imagem.
d) O operador parênteses com 2 argumentos (· , ·) irá acessar o pixel (*x*,*y*) com o centro localizado em *centro*(*l*,*c*) e irá retornar *backg* se acessar um pixel inexistente.

O programa 5 exemplifica o uso dessa classe.

	programa	saída
1	`//imgxyb.cpp 2024`	
2	`#include "procimagem.h"`	
3	`int main() {`	
4	`  ImgXyb<uchar> a(3,3);`	
5	`  a << 1,2,3,`	
6	`       4,5,6,`	
7	`       7,8,9;`	
8	`  a.centro(1,1); a.backg=255;`	
9	`  printf("a(-1,-1)=%d\n",a(-1,-1));`	`a(-1,-1)=7`
10	`  printf("a(-1,+1)=%d\n",a(-1,+1));`	`a(-1,+1)=1`
11	`  printf("minx=%d maxx=%d miny=%d maxy=%d\n",`	`minx=-1 maxx=1 miny=-1 maxy=1`
12	`         a.minx, a.maxx, a.miny, a.maxy);`	
13	`  printf("a(-2,-1)=%d\n",a(-2,-1));`	`a(-2,-1)=255`
14	`}`	

Programa 5: Exemplo de uso da classe *ImgXyb*.

Nas linhas 4-7, o programa criou uma imagem 3×3 e o preencheu com valores de 1 a 9 (figura 12a). Na linha 8, o pixel *l*=1, *c*=1 foi definido como centro da imagem através do comando "a.centro(1,1)". Além disso, foi definido que background é 255 (isto é, os pixels fora do domínio são 255). Depois destes comandos, a *ImgXyb a* fica o acesso *a*(*x*,*y*) como mostrado na figura 12b.

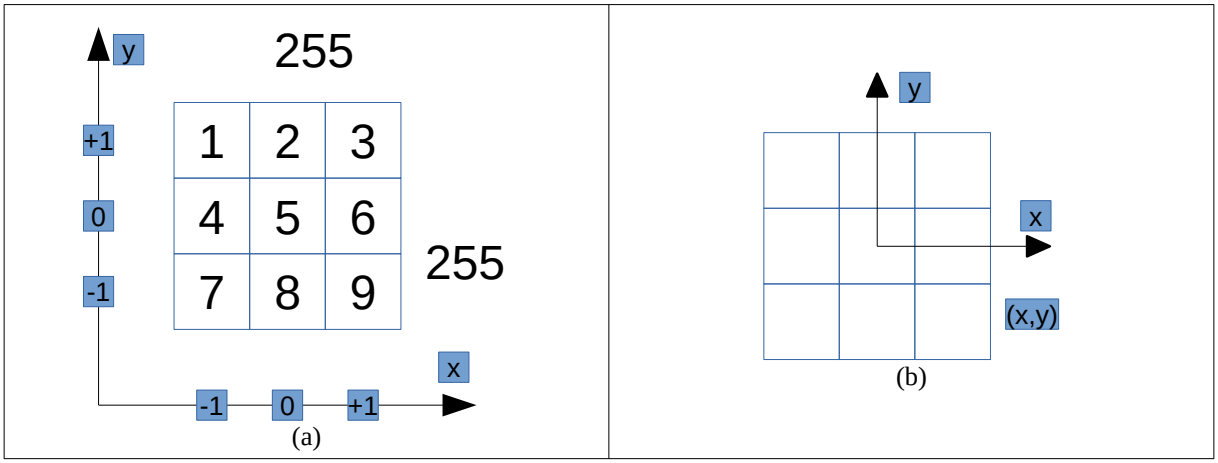


Figura 12: (a) ImgXyb *a* do programa 5. (b) Sistema de coordenadas "xy-centralizado".

Assim, a(-1,-1) irá acessar "7" e a(-1,+1) irá acessar "1". Os membros *minx*, *maxx*, *miny* e *maxy* serão respectivamente -1, +1, -1 e +1, indicando os limites do domínio da imagem. Se acessar qualquer pixel fora do domínio, por exemplo a(-2,-1), a classe retorna a cor de background 255. É possível especificar `ImgXyb<Vec3b>`, `ImgXyb<float>`, etc.

Além da classe ImgXyb, Cekeikon possui as seguintes classes:

- *ImgLcb* e *ImgXyb* - Acessa imagem no modo lc- ou xy-centralizado com centro definido pelo comando "centro". Se acessar pixel fora do domínio, devolve cor definido pelo backg (semelhante a BORDER_CONSTANT do OpenCV, iiiiii|abcdefgh|iiiiiii).
- *ImgLcx* e *ImgXyx* - Modo lc- ou xy-centralizado. Se acessar pixel fora do domínio, retorna o valor do pixel espacialmente mais próximo dentro do domínio (semelhante a BORDER_REPLICATE do OpenCV, aaaaaa|abcdefgh|hhhhhhh).
- *ImgLce* e *ImgXye* - Modo lc- ou xy-centralizado. Se acessar pixel fora do domínio, gera erro.
- *ImgLcr* e *ImgXyr* - Modo (l,c) ou (x,y) centralizado. Funciona como BORDER_WRAP do OpenCV (cdefgh|abcdefgh|abcdefg).

Nota: A lição de casa da primeira aula (eliminar ruído da orelha do Mickey) e o exercício 1 da apostila "comp-con-ead" (pintar a área externa branca do Mickey) teriam ficado muito mais simples se tivéssemos usado *ImgLcb*, pois esta classe permite especificar a cor de um pixel fora do domínio da imagem. Experimente resolver esse exercício usando *ImgLcb*.

2.3 Implementação

Finalmente, estamos prontos para escrever o programa de rotação. O programa 6 rotaciona a imagem em torno do seu centro por um ângulo especificado. Executando:

```
>rotacao lennag.jpg rotacao.jpg 30
```

obtemos a imagem mostrada na figura 13 esquerda.

A linha 8 lê o ângulo de rotação em graus e o converte para radianos. As linhas 10-11 calculam seno e cosseno do argumento.

A linha 13 lê a imagem de entrada como *ImgXyb* para acessá-lo com coordenadas cartesianas (x, y). A linha 14 define o centro da imagem como o centro do sistema de coordenadas e define que *background* da imagem é branco. As linhas 15-16 fazem o mesmo para a imagem de saída.

As linhas 18-23 percorrem a imagem de saída *b* calculando a cor de cada um dos pixels. As linhas 20-21 aplicam a equação da figura 11 para converter as coordenadas (x_b, y_b) da imagem de saída *b* para coordenadas (x_a, y_a) da imagem de entrada *a*. A linha 22 aplica interpolação vizinho mais próximo.

```
1  //rotacao.cpp 2024
2  #include "procimagem.h"
3
4  inline double deg2rad(double x)
5  { return (x/180.0)*(M_PI); }
6
7  int main(int argc, char** argv)
8  { if (argc!=4) {
9      printf("rotacao ent.pgm sai.pgm graus\n");
10     erro("Erro: Numero de argumentos invalido");
11   }
12   double graus; sscanf(argv[3],"%lf",&graus);
13   double radianos=deg2rad(graus);
14   double co=cos(radianos);
15   double se=sin(radianos);
16
17   ImgXyb<uchar> a=imread(argv[1],0); if (a.total()==0) erro("Erro leitura");
18   a.centro(a.rows/2,a.cols/2); a.backg=255;
19   ImgXyb<uchar> b(a.rows,a.cols);
20   b.centro(b.rows/2,b.cols/2); b.backg=255;
21
22   for (int xb=b.minx; xb<=b.maxx; xb++)
23     for (int yb=b.miny; yb<=b.maxy; yb++) {
24       int xa=cvRound(xb*co+yb*se);
25       int ya=cvRound(-xb*se+yb*co);
26       b(xb,yb)=a(xa,ya);
27     }
28   imwrite(argv[2],b);
29 }
```

Programa 6: Programa que rotaciona imagem, usando interpolação vizinho mais próximo C++/OpenCV.

lennag.jpg

rotacao.jpg

Figura 13: Entrada e saída do programa *rotacao.cpp*.

Exercício: Reescreva o program *rotacao.cpp* mudando classe *ImgXyb* para *ImgXyr* e *ImgXyx*. Execute os programas modificados e verifique a diferença no tratamento dos pixels fora do domínio.

Exercício: Reescreva o program *rotacao.cpp* para efetuar a interpolação bilinear (em vez da interpolação vizinho mais próximo).

Exercício: Reescreva o programa *rotacao.cpp* sem usar a classe *ImgXy?* ou *ImgLc?*. Não pode usar a função pronta do OpenCV que efetua rotação.

Exercício: Traduza o program *rotacao.cpp* para Python.

2.4 Função da OpenCV que efetua rotação

Implementamos manualmente o programa *rotacao.cpp* para dar explicação didática do seu funcionamento. Porém, como das outras vezes, há uma função pronta do OpenCV que efetua a rotação. Na verdade, não é uma função mas uma sequência de duas funções que, juntas, efetuam a rotação.

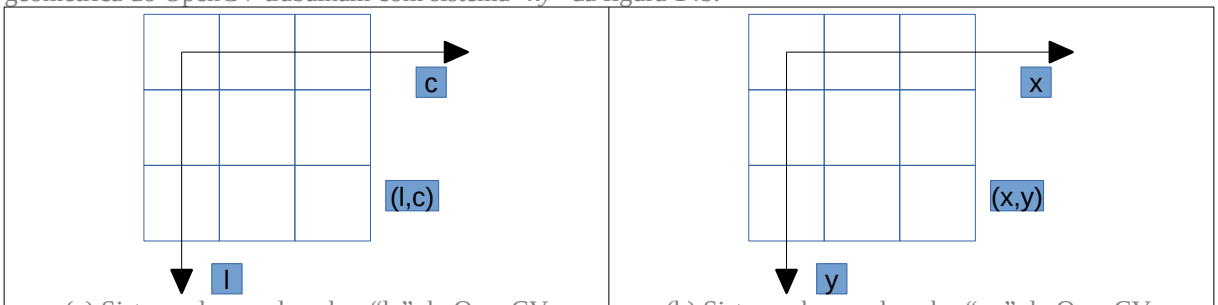
getRotationMatrix2D: Calcula a matriz afim (2×3) que executa a rotação 2D desejada.

```
C++: Mat getRotationMatrix2D(Point2f center, double angle, double scale)
Python: cv2.getRotationMatrix2D(center, angle, scale) → retval
```

warpAffine: Aplica uma transformação afim a uma imagem.

```
C++: void warpAffine(InputArray src, OutputArray dst, InputArray M, Size dsize,
   int flags=INTER_LINEAR, int borderMode=BORDER_CONSTANT, const Scalar& borderValue=Scalar())
Python: cv2.warpAffine(src, M, dsize[, dst[, flags[, borderMode[, borderValue ] ] ] ]) → dst
```

Nota: Para evitar confusão, destaco na figura 14 os dois sistemas de coordenadas usados pela OpenCV. Em algumas funções, OpenCV usa sistema "*lc*" e em outras usa sistema "*xy*" (com *y* de cima para baixo). Não confunda com o sistema *xy*-centralizado (com *y* para cima) que usamos na classe *ImgXyb*. As funções de transformação geométrica do OpenCV trabalham com sistema "*xy*" da figura 14b.



(a) Sistema de coordenadas "lc" do OpenCV (b) Sistema de coordenadas "xy" do OpenCV.

Figura 14: Algumas funções do OpenCV usam o sistema de coordenadas "*lc*" e outras usam o sistema de coordenadas "*xy*". Em transformações geométricas, o sistema utilizado é "*xy*".

```
 1 // rotacao_cv.cpp 2024
 2 #include "procimagem.h"
 3 int main() {
 4   Mat_<uchar> ent=imread("lennag.jpg",0);
 5   Mat_<uchar> sai;
 6   Mat_<double> m=getRotationMatrix2D(Point2f(ent.cols/2,ent.rows/2), 30, 1);
 7   cout << m << endl;
 8   warpAffine(ent, sai, m, ent.size(), INTER_LINEAR, BORDER_CONSTANT, Scalar(255));
 9   imwrite("rotacao_cv.jpg",sai);
10 }
```

Programa 7: Rotação usando funções do OpenCV.

O programa 7 (rotacao_cv.cpp) rotaciona a imagem *lennag.jpg* de 30 graus em sentido anti-horário em torno do seu centro. Na linha 6, a função *getRotationMatrix2D* cria uma matriz afim 2×3 que converte coordenada (x_a, y_a) da imagem de entrada *a* para coordenada (x_b, y_b) da imagem de saída *b*.

Cuidado: Os argumentos de Point2f são (x,y) e não (l,c).

A linha 7 do programa imprime a matriz *m* que faz transformação direta (x_a, y_a) para (x_b, y_b):

`[0.866, 0.499, -93.702;` `-0.499, 0.866, 162.29]`	$\begin{bmatrix} x_b \\ y_b \end{bmatrix} = \begin{bmatrix} 0.866 & 0.499 & -93.702 \\ -0.499 & 0.866 & 162.29 \end{bmatrix} \begin{bmatrix} x_a \\ y_a \\ 1 \end{bmatrix}$

Porém, como vimos antes, para fazer rotação é necessária a transformação inversa, de (x_b, y_b) para (x_a, y_a). Assim, muito provavelmente, OpenCV deve calcula internamente a matriz inversa m^{-1} para fazer a rotação. A matriz inversa de uma transformação afim pode ser calculada pela função *invertAffineTransform*.

Depois, a linha 8 efetua a rotaçao, usando interpolação bilinear. Executando o programa 7 (rotacao_cv.cpp), obtemos uma saída muito semelhante à mostrada na figura 13.

Exercício: Ao fazer a rotação, as esquinas da imagem desaparecem, como mostra a figura 13. Modifique o programa 7 para que nenhuma parte da imagem de entrada seja perdida ao fazer rotação.

Exercício: Altere os parêmetros *flags*, *borderMode* e *borderValue* do programa 7 para verificar os efeitos.

3. Transformações geométricas 2D

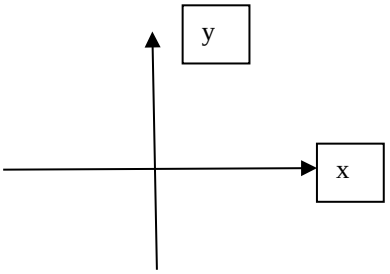
Como vimos nas seções anteriores, para fazer qualquer transformação geométrica, precisa saber calcular, para cada pixel (x_B, y_B) da imagem de saída *B*, a coordenada correspondente (x_A, y_A) da imagem de entrada *A*. Depois disso, basta aplicarmos uma das técnicas de interpolação que já vimos (vizinho mais próximo, bilinear, bicúbico, Lanczos e por área). Vamos ver agora algumas classes de transformações.

3.1 Transformação linear (matriz 2×2)

Representando os pontos no plano por vetores 2×1, uma matriz 2×2 representa a transformação linear. Matriz 2×2 consegue representar rotação (em torno do centro do sistema de coordenadas), cisalhamento (shearing), reflexão e mudança de escala. Não consegue representar translação nem rotação em torno de um ponto arbitrário. Exemplos:

Rotação (em torno do ponto (0,0)):

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} x_A \\ y_A \end{bmatrix}, \; c=\cos(\alpha) \text{ e } s=\sin(\alpha)$$



Cisalhamento (shearing):

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_A \\ y_A \end{bmatrix}$$

Reflexão em torno do eixo *x* (multiplique a matriz por -1 para reflexão em torno do eixo *y*).

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_A \\ y_A \end{bmatrix}$$

Mudança de escala:

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} e_x & 0 \\ 0 & e_y \end{bmatrix} \begin{bmatrix} x_A \\ y_A \end{bmatrix}$$

Composição de transformações pode ser obtida multiplicando as matrizes de transformação. A transformação inversa é calculada pela inversa da matriz.

3.2 Transformação afim (matriz 2×3)

Transformações afins são representadas por matrizes de transformação 2×3 que incluem (além de todas das transformações lineares 2×2) a translação e rotação em torno de um ponto arbitrário.

A matriz de translação do ponto (x_A, y_A) por vetor (t_x, t_y) é:

$$\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix} \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix}$$

Já vimos como efetuar rotação de uma imagem em torno de um ponto arbitrário usando matriz afim.

A função *getAffineTransform* de OpenCV determina a matriz de transformação afim genérica 2×3 que mapeia um paralelogramo em outro paralelogramo a partir de 3 pares de pontos correspondentes, mantendo o paralelismo das retas. O programa 8 ilustra o uso dessa função para fazer cisalhamento.

As linhas 6 e 11 estão dizendo que o ponto de entrada (0,0) deve ser mapeado no ponto de saída (200, 100). Similarmente, as linhas 7 e 12 e as linhas 8 e 13 criam outras duas correspondências de pontos entrada-saída. Executando o programa, obtemos a imagem Lenna com cisalhamento.

```
 1  //shear.cpp pos2016
 2  #include <cekeikon.h>
 3
 4  int main()
 5  { Mat_<FLT> src = (Mat_<FLT>(3,2) <<
 6      0,0,
 7      0,511,
 8      511,511);
 9    cout << src << endl;
10    Mat_<FLT> dst = (Mat_<FLT>(3,2) <<
11      200,100,
12      100,400,
13      400,400);
14    cout << dst << endl;
15    Mat_<FLT> m=getAffineTransform(src,dst);
16    cout << m << endl;
17
18    Mat_<GRY> a; le(a,"lenna.jpg");
19    Mat_<GRY> b;
20    warpAffine(a,b,m,a.size(),INTER_LINEAR,BORDER_WRAP);
21    imp(b,"afim.png");
22  }
```

Programa 8: Exemplo de uso de *getAffineTransform* para fazer cisalhamento.

Saída:

```
[0.58708417, -0.19569471, 200;
 0, 0.58708417, 100]
```

A função *invertAffineTransform* do OpenCV calcula a transformação inversa de uma matriz afim 2×3. Evidentemente, não é possível compor várias transformações afins simplesmente fazendo multiplicação matricial pois não dá para multiplicar matricialmente duas matrizes 2×3. Usando matrizes 3×3, poderemos compor transformações multiplicando matrizes e calcular a transformação inversa invertendo a matriz.

3.3 Coordenadas homogêneas 2D

A transformação afim com matriz 2×3 não consegue representar transformação em perspectiva. Além disso, não há uma forma fácil de compor várias transformações e calcular inversa. Para superar essas dificuldades, podemos utilizar matriz 3×3 com coordenadas homogêneas.

Coordenadas homogêneas (ou coordenadas projetivas, abreviado CHs) é um sistema de coordenadas usada na geometria projetiva (figura 15). Um ponto (x, y) em R^2 é representado em CHs utilizando 3 números (x, y, w). O ponto (x, y, w) em CHs representa o ponto (x/w, y/w) em R^2.

Por exemplo, todos pontos $(x, y, w) = (4, 6, 2)$, $(6, 9, 3)$ e $(1, 1.5, 0.5)$ em CHs representam o mesmo ponto $(2, 3)$ de R^2. Quando $w=1$, dizemos que a representação está normalizada. Exemplo: a representação normalizada do ponto $(4, 6, 2)$ em CHs é $(2, 3, 1)$. Para converter um ponto (x, y) para CHs, basta acrescentar o número um na terceira coordenada $(x, y, 1)$.

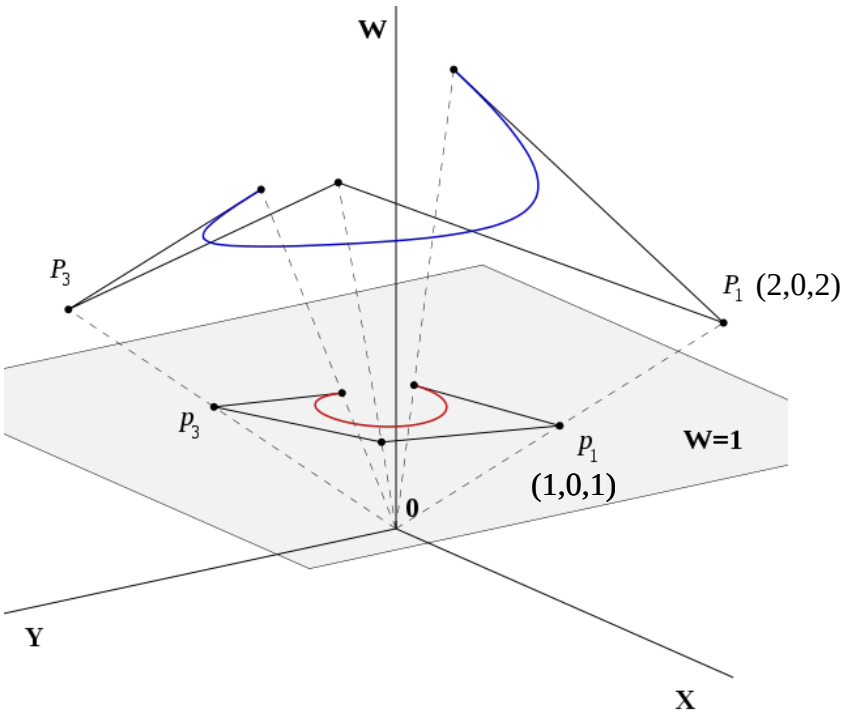


Figura 15: Coordenadas homogêneas (retirada de Wikipedia).

3.4 Transformação perspectiva (matriz 3×3)

Usando sistema de coordenadas homogêneas juntamente com matrizes 3×3, podemos efetuar rotação, translação, reflexão, mudança de escala, transformação afim e transformação em perspectiva.

A composição das transformações é calculada pela multiplicação matricial. A transformação inversa é dada pela inversa da matriz.

Matriz de rotação em coordenadas homogêneas R_α que rotaciona α em torno do ponto (0,0):

$$\begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} = \begin{bmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} \qquad \text{onde } c=\cos(\alpha) \text{ e } s=\sin(\alpha)$$

Matriz de translação em coordenadas homogêneas T_t que translada para vetor $t = (t_x, t_y)$.

$$\begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix}$$

Matriz de mudança de escala em coordenadas homogêneas E_e que muda escala $e = (e_x, e_y)$.

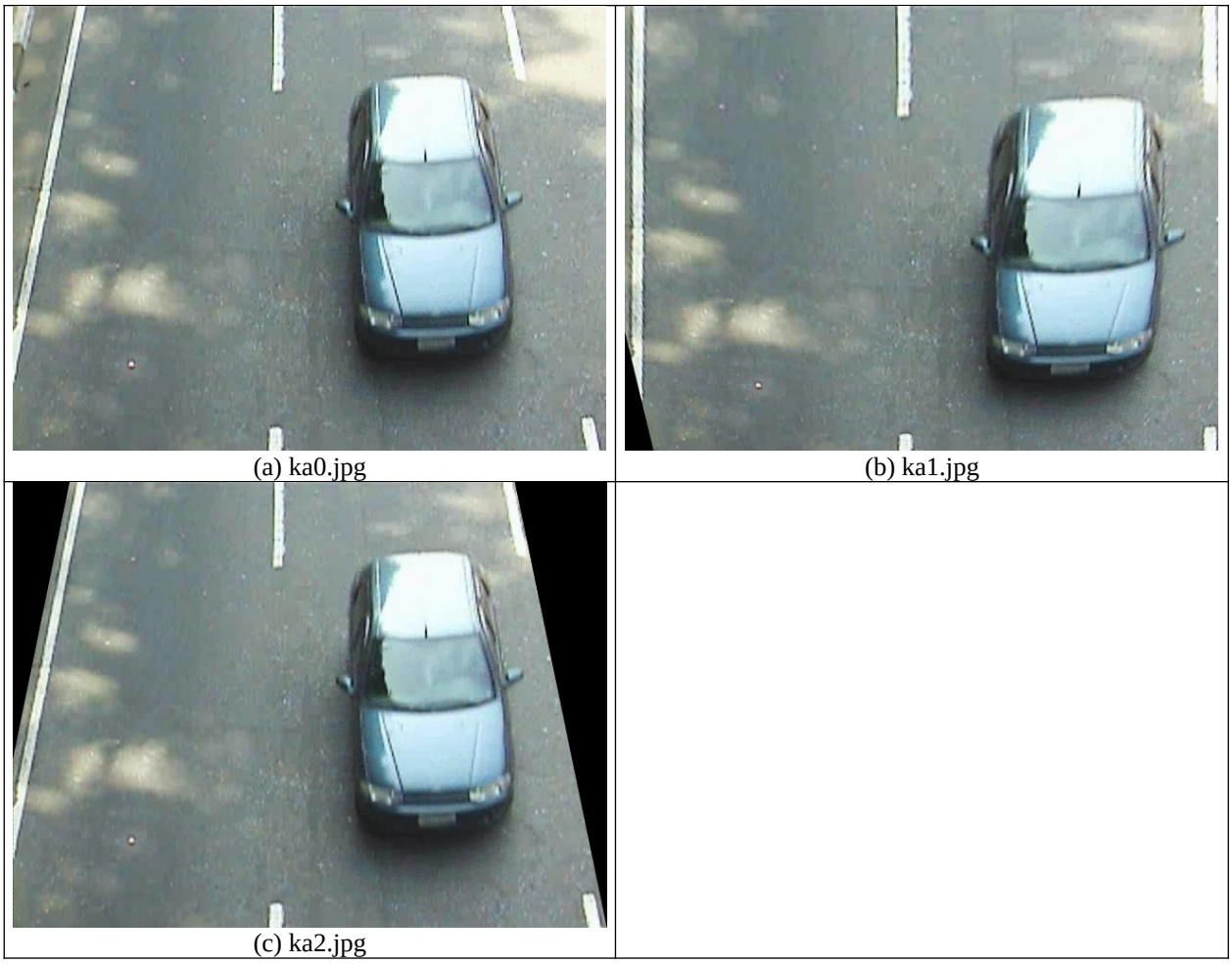
$$\begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} = \begin{bmatrix} e_x & 0 & 0 \\ 0 & e_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix}$$

As matrizes podem ser multiplicadas para obter transformação composta. Por exemplo, a matriz de rotação de α graus em torno do ponto $c = (c_x, c_y)$ é:

$T_c \cdot R_\alpha \cdot T_{-c}$.

3.5 Correção de perspectiva

Digamos que queremos calcular velocidades dos carros que trafegam numa estrada. Para isso, instalamos uma câmera no alto de um pórtico. A figura 16a mostra uma imagem adquirida. O problema é que os pixels possuem dimensões diferentes. Um pixel na parte superior da imagem representa uma área maior na rodovia do que um pixel na parte inferior. Assim, a velocidade do carro não pode ser calculada somente a partir de quantos pixels o carro se moveu entre *n* quadros consecutivos. Uma forma de resolver este problema seria fazer correção de perspectiva: converter figura 16a para 16b. Na figura 16b, todos os pixels possuem dimensões semelhantes, de forma que é possível calcular a velocidade dos carros a partir da informação de quantos pixels o carro se moveu entre *n* quadros consecutivos.



(a) ka0.jpg

(b) ka1.jpg

(c) ka2.jpg

Figura 16: Corrigindo a distorção em perspectiva da imagem (a), obtemos a imagem (b). Fazendo a transformada inversa da imagem (b), obtemos de volta a imagem original (c).

O programa 9 faz esta correção. As linhas 5-9 listam 4 coordenadas da imagem de entrada e as linhas 10-14 listam as 4 coordenadas correspondentes da imagem de saída (para onde gostaríamos que os 4 pontos de entrada fossem mapeados). A partir destas duas sequências de pontos, o função *getPerspectiveTransform* da linha 15 constrói matriz *m* (3×3) que efetua a transformação desejada. A linha 16 imprime esta matriz.

Para verificar que a matriz *m* efetua a transformação desejada, testei se o ponto (-22,479) seria mapeada em (14,279) como desejamos. Para isso, coloquei o ponto (-22, 479) em CHs e o armazenei em *v*=(-22, 479, 1). Depois, pedi para calcular *m***v*, resultando *w*=(19.7, 675, 1.41). Normalizando CHs, obtemos (19.7/1.41, 675/1.41)=(14, 479), que era exatamente a saída desejada (linha 13 do programa 9). Isto confirma que a matriz *m* está efetuando a transformação desejada.

A linha 26 efetua a correção de perspectiva e imprime a imagem resultante como *ka1.jpg* (figura 16). As linhas 30-32 fazem a transformação inversa, voltando a obter a imagem em perspectiva e o imprime em *ka2.jpg*.

```
 1 //pers.cpp grad-2018
 2 #include "procimagem.h"
 3
 4 int main() {
 5   Mat_<float> src = (Mat_<float>(4,2) <<
 6     73,0,
 7     533,0,
 8     -22,479,
 9     629,479);
10   Mat_<float> dst = (Mat_<float>(4,2) <<
11     16,0,
12     630,0,
13     14,479,
14     630,479);
15   Mat_<double> m=getPerspectiveTransform(src,dst);
16   cout << m << endl;
17
18   //Verifica se a transformacao esta fazendo o que queremos
19   Mat_<double> v=(Mat_<double>(3,1) << -22,479,1);
20   Mat_<double> w=m*v;
21   cout << w << endl;
22   cout << w(0)/w(2) << " " << w(1)/w(2) << endl;
23
24   //Corrige a perspectiva
25   Mat_<Vec3b> a=imread("ka0.jpg",1);
26   Mat_<Vec3b> b;
27   warpPerspective(a,b,m,a.size());
28   imwrite("ka1.jpg",b);
29
30   //Refaz a perspectiva
31   m=m.inv();
32   warpPerspective(b,a,m,a.size());
33   imwrite("ka2.jpg",b);
34 }
```

Programa 9: Correção de perspectiva.

Saída:

```
[1.334782608695661, 0.2725533679355095, -81.43913043478391;
  2.040034807748725e-15, 1.410622529644279, -7.602807272633072e-13;
  7.426784881525705e-18, 0.0008572495399671701, 1]

[19.74871541502058;
  675.6881916996088;
  1.410622529644274]

14 479
```

Exercício: Modifique o programa 9 para obter programa que converte quadrado1.png em quadrado1b.png.

Exercício: Modifique o programa 9 para obter programa que converte quadrado2.png em quadrado2b.png.

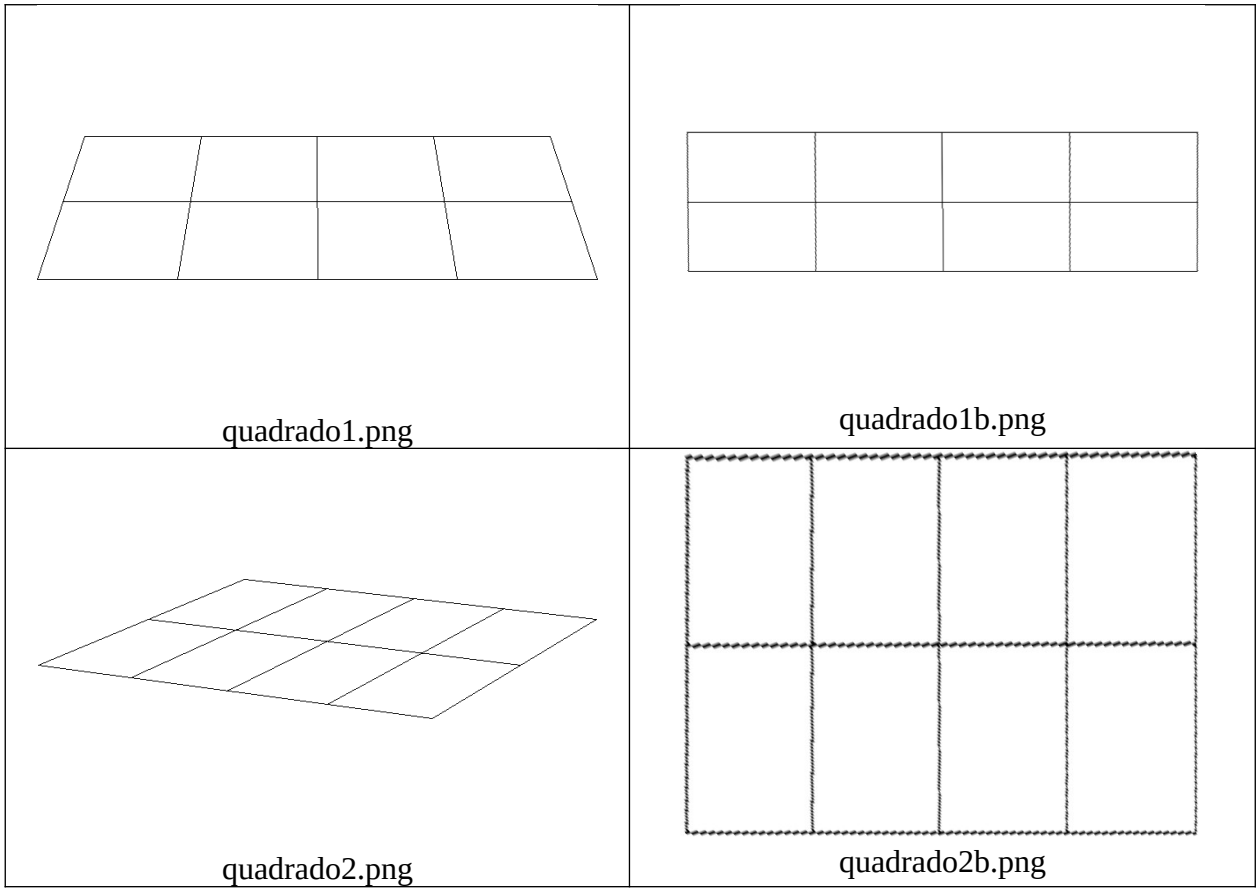


Figura 17: Outros exemplos de transformação em perspectiva.

[PSI5790-2025 aula 3 lição de casa #1 (de 2). Vale 5,0] Corrija a deformação em perspectiva do tabuleiro de xadrez abaixo, gerando uma imagem onde cada casa do tabuleiro é um quadrado alinhado aos eixos do sistema de coordenadas. Consequentemente, o tabuleiro todo será um retângulo alinhado aos eixos do sistema de coordenadas.
Nota: Você não precisa determinar automaticamente as esquinas do tabuleiro. Pode colocar no seu programa, manualmente, as suas coordenadas.

calib_result.jpg

Exemplos de imagens com correção de distorção em perspectiva.

Anexo A: Curiosidades

Transformações geométricas podem ser usadas para outras finalidades. Por exemplo, é possível corrigir distorção da lente e fazer “morphing”. As figuras 18, 19 e 20 mostram exemplos de outras transformações geométricas.

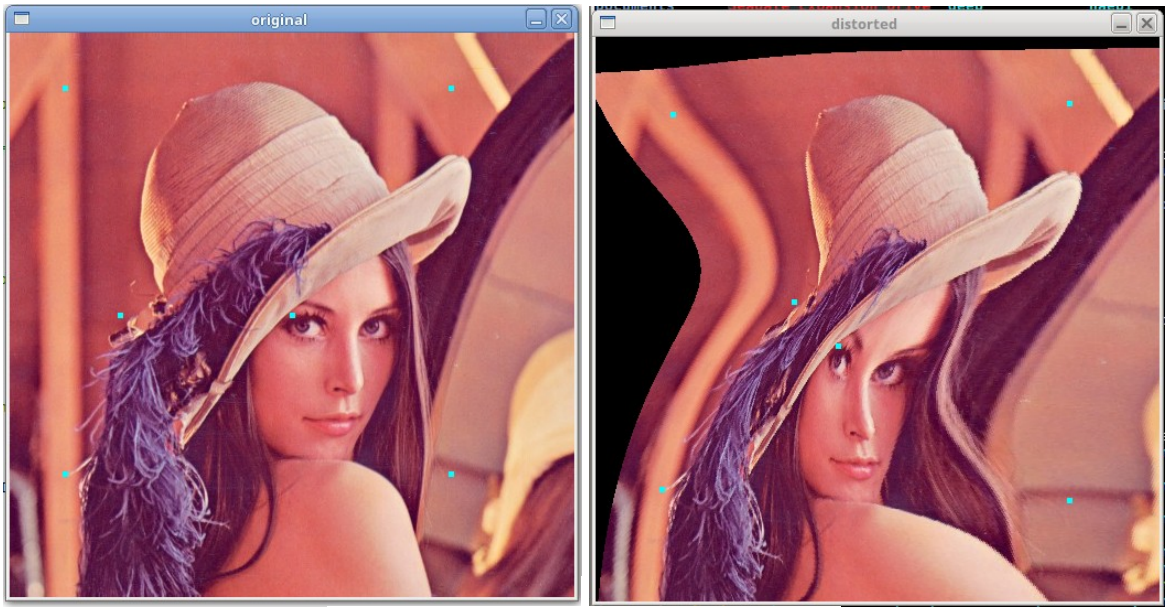


Figura 18: Distorção “thin plate spline”.

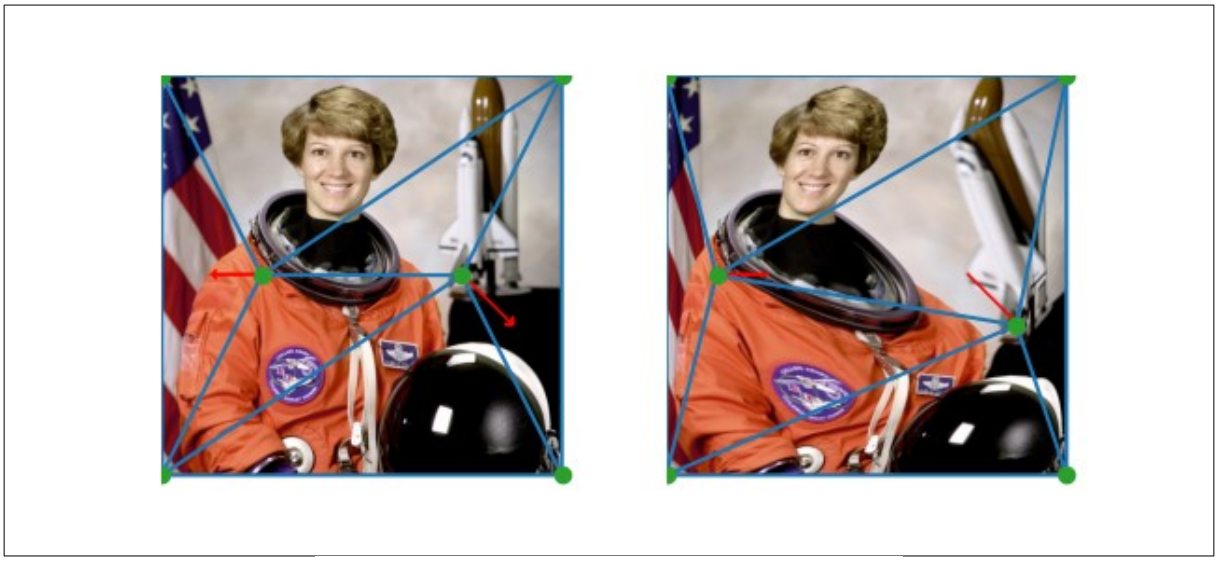


Figura 19: Transformação “piecewise affine”.

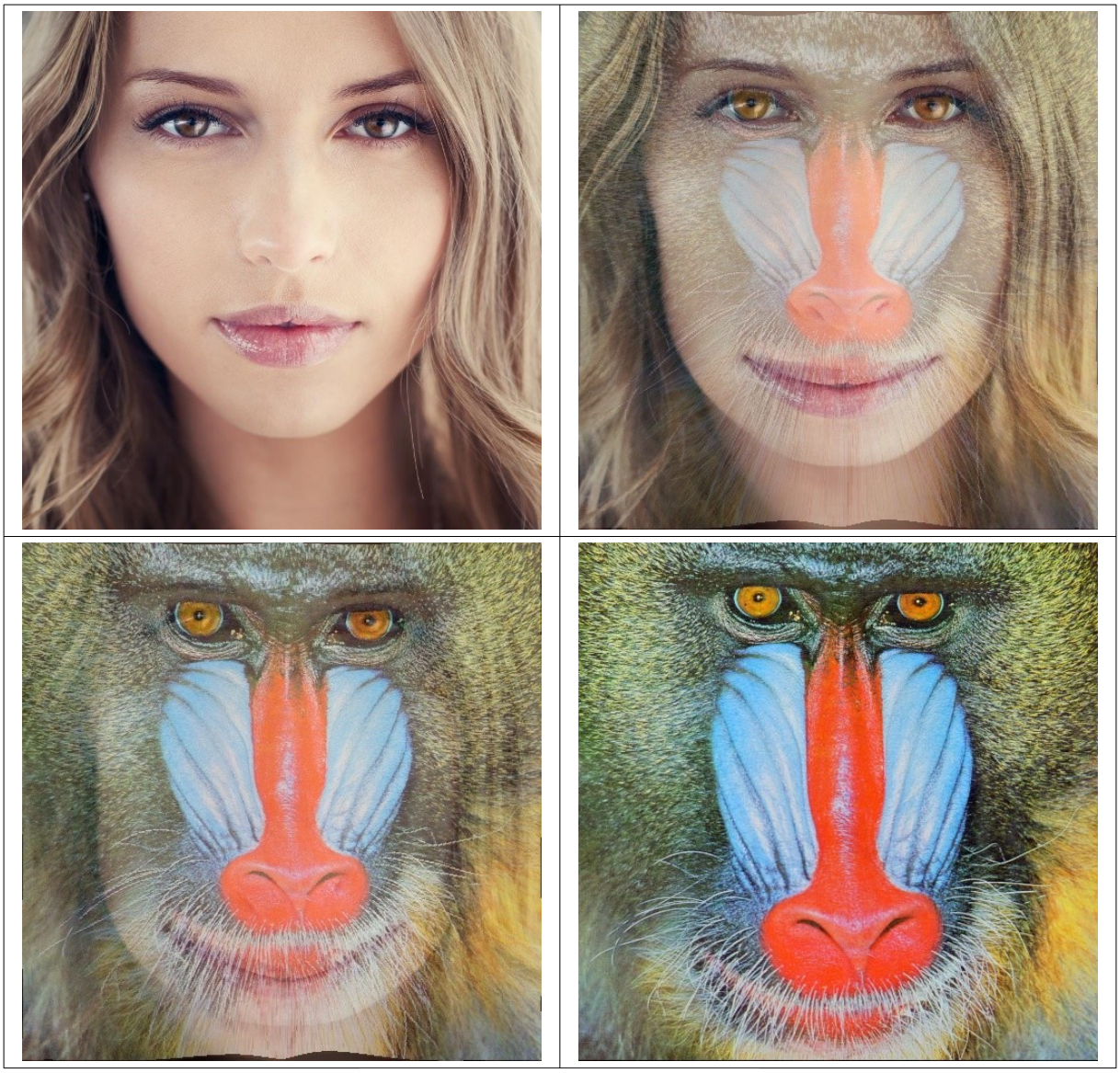


Figura 20: Exemplo de *morphing*.

Anexo B: Pintar fundo de Mickey de azul usando ImgLcb.

Na aula “compcon-ead”, desenvolvemos algoritmos para pintar partes de Mickey de azul usando fila/pilha. Vimos que esses algoritmos não funcionam para pintar o fundo da imagem de azul. Tivemos que testar se um pixel está dentro ou fora do domínio da imagem. Trocando Mat_<COR> por ImgLcb<COR> e definindo background como preto no programa pintafila1.cpp, o programa passa a funcionar sem alterar mais nada.

```
1  //imglcb.cpp - grad2023
2  #include <cekeikon.h>
3  #include <queue>
4
5  ImgLcb<COR> pintaAzul(ImgLcb<COR> a, int ls,
6  int cs) {
7    ImgLcb<COR> b=a.clone();
8    queue<int> q;
9    q.push(ls); q.push(cs); //(1)
10   while (!q.empty()) { //(2)
11     int l=q.front(); q.pop(); //(3)
12     int c=q.front(); q.pop(); //(3)
13     if (b(l,c)==COR(255,255,255)) { //(4)
14       b(l,c)=COR(255,0,0); //(5)
15       q.push(l-1); q.push(c); //6-acima
16       q.push(l+1); q.push(c); //6-abaixo
17       q.push(l); q.push(c+1); //6-direita
18       q.push(l); q.push(c-1); //6-esq
19     }
20   }
21   return b;
22 }
23
24 int main() {
25   ImgLcb<COR> a; le(a,"mickey_reduz.bmp");
26   a.backg=COR(0,0,0);
27   ImgLcb<COR> b=pintaAzul(a,268,292);
28   imp(b,"imglcb.png");
29 }
```

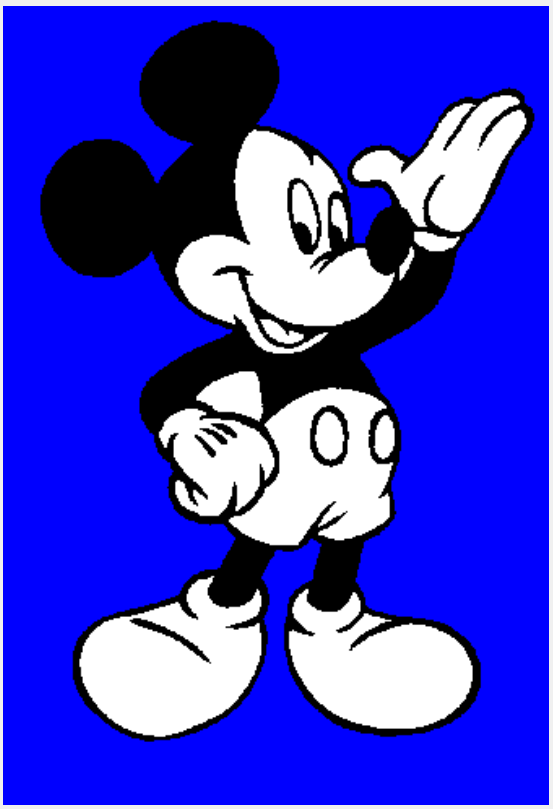


Figura: Usando ImgLcb, é possível pintar o fundo do Mickey, praticamente sem alterar o programa “pintafila1.cpp” da apostila “compcon-ead”.

Referências:

[Pratt1991] William K. Pratt, “Digital Image Processing,” 2nd ed., John Wiley & Sons, 1991.

[wikiBicubic] https://en.wikipedia.org/wiki/Bicubic_interpolation

[wikiAliasing] https://en.wikipedia.org/wiki/Aliasing

[scikitReduce] https://scikit-image.org/docs/0.7.0/api/skimage.transform.pyramids.html#pyramid-reduce

[opencvPyramids] https://docs.opencv.org/2.4/doc/tutorials/imgproc/pyramids/pyramids.html

[PSI5790-2025 aula 3 parte 1 – fim]